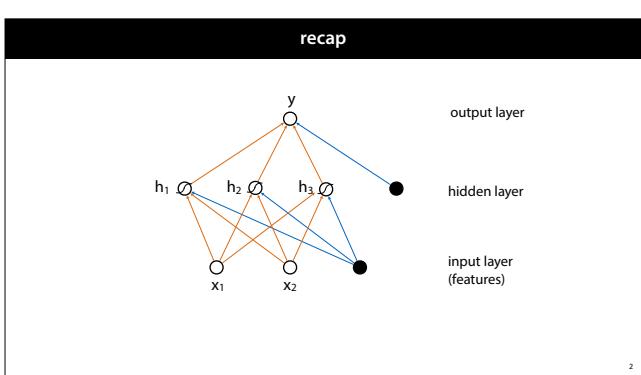


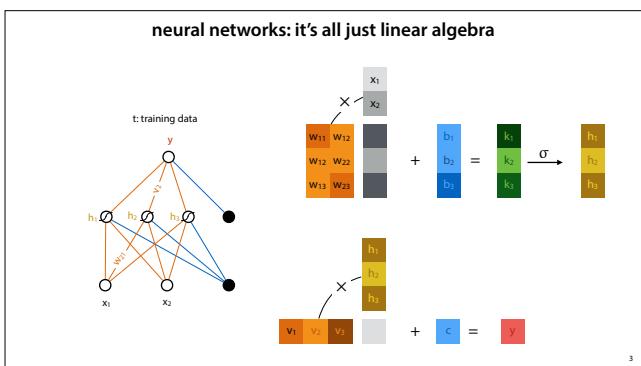
This is the first of several lectures that deal with deep learning. Deep learning evolved out of neural networks, but it's slightly more than just the business of training very large and deep neural nets. We'll discuss exactly what makes a deep learning model at the end of the lecture.



For now, we'll pick up the discussion about neural networks from the last lecture, and develop the idea further.

Here's how we defined a neural network last time around. A neural network is a model described by a graph: each node represents a scalar value. The value of each node is computed from the incoming edges by multiplying the weight on the edge by the value of the node it connects to.

We train the model by tuning the weights. Every **orange** and **blue** line in this picture represents one of those weights. The feedforward network, seen here, is the simplest way of wiring up a neural net, but we will see



In addition to the graph perspective, there's another perspective that can greatly simplify things. Most of the operations used in neural networks can also be written as matrix multiplications.

$$f(\mathbf{x}) = \mathbf{V} \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

As you can see, this greatly simplifies our notation. It also allows for very efficient *implementation* of neural networks, since matrix multiplication can be implemented very efficiently (especially on a GPU).

This is what we will discuss today: how to simplify the basic idea of neural networks into a very powerful and flexible framework for creating highly complex machine learning models.

**deep learning**

---

**Part 1:** Deep learning systems (aka autodiff engines)  
tensors, matrix calculus, backpropagation revisited

**Part 2:** Backpropagation revisited  
Multivariate chain rule, calculus over tensors

**Part 3:** Convolutions  
aka ConvNets or CNNs

**Part 4:** Making it work  
vanishing gradients, minibatches, optimizers, regularization

for more detail, see [dlvu.github.io](https://dlvu.github.io)

5

In [this video](#), we'll look at the general layout of deep learning frameworks. Specifically, how these systems allow us to define complex models in such a way that they can be efficiently computed and that we don't have to implement backpropagation ourselves. A large part of this, is expressing everything we do in the language of *tensors*.

In the [second video](#), we'll look specifically at how we can implement backpropagation in such tensor settings. We've seen a scalar version of the algorithm already, where we work out the derivatives of the parameters one by one, but when we want to implement this efficiently, using tensor operations, we need to take a few more things into account.

Deep learning works best when we don't use fully connected layers, but when we tailor the architecture of our network to our task. In the [third video](#), we'll look at the first type of layer that allows us to do this: the convolution. This layer is particularly suited to image data.

And finally, to do deep learning efficiently, we need to know a number of tricks and tools. We'll run through the most important ones briefly in the [last video](#).

We don't have time in this lecture to go into all the details. If you are doing your project on a deep learning topic, and you need to know more, you can have a look at the materials for our Deep Learning course in the master. They discuss the same subjects, but in more detail, with more examples (in particular lectures 2, 3, and 4).

**aims**

---

**Scalar backpropagation -> Tensor backpropagation**  
Tensor operations (i.e. matrix multiplication) are easy to parallelise and run well on GPUs

**Functions with multiple inputs and outputs**

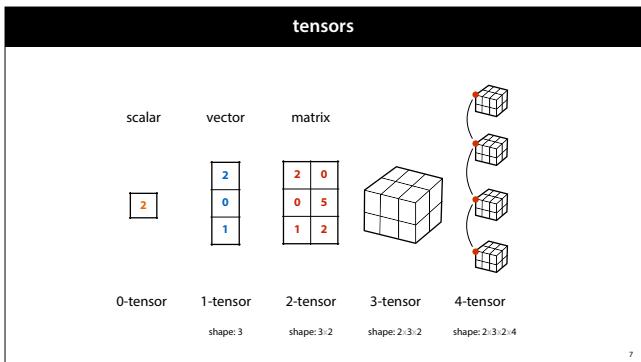
**Backpropagation over any computation (next video)**  
No matter how complex

Ingredients: **tensors** and **functions over tensors**

6

These are our aims for the first two videos. In order to scale up the basic principle of backpropagation on neural networks, we want to move from operations on scalars to the linear algebra view: everything is a tensor and all operations (including this in the backpropagation step) are tensor operations. Including the backpropagation algorithm

We will first define two ingredients: tensors, and functions over tensors.



The main ingredient in the language of deep learning is the **tensor**. For our purposes, a tensor is nothing more than a straightforward generalisation of vectors and matrices to higher dimensionalities. A tensor is collection of numbers arranged in a array. The **rank** of the tensor, is the number of dimensions along which the values change.

A vector is a rank 1 tensor: a one-dimensional array. Its shape can be described by one integer: how many numbers there are arranged along one dimension.

A matrix is a rank 2 tensor: a two dimensional array. Its shape is described by two integers: how many rows it has and how many columns.

In this logic, we often think of a asingle number (a scalar) as a rank 0 tensor.

Extending this idea, we can create tensors of rank 3, 4 and higher. Above rank three, they're not easy to visualize, but you can think of a rank 4 tensor as a collection of rank-3 tensors, arranged along an extra dimension (just like a matrix is a collection of vectors arranged along an extra dimension).

If you've done the first worksheet, you'll already know the tensor as the basic data structure of numpy. There it is more often called a **multidimensional array**.

The idea of deep learning frameworks is that we can deal with any data, so long as it's represented as a tensor. Let's look at some examples of how data is represented in tensor form.

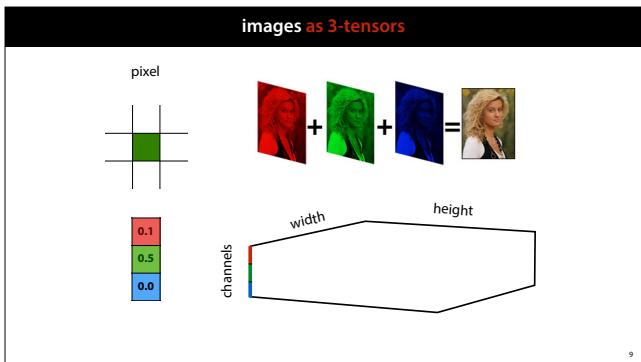
**data as tensor**

	$X =$ <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>181</td><td>46</td><td>male</td></tr> <tr><td>181</td><td>50</td><td>male</td></tr> <tr><td>166</td><td>44</td><td>female</td></tr> <tr><td>171</td><td>38</td><td>female</td></tr> <tr><td>152</td><td>36</td><td>female</td></tr> <tr><td>156</td><td>40</td><td>female</td></tr> <tr><td>167</td><td>40</td><td>female</td></tr> <tr><td>170</td><td>45</td><td>male</td></tr> <tr><td>178</td><td>50</td><td>male</td></tr> <tr><td>191</td><td>50</td><td>male</td></tr> </table>	181	46	male	181	50	male	166	44	female	171	38	female	152	36	female	156	40	female	167	40	female	170	45	male	178	50	male	191	50	male	$y =$ <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td></tr> <tr><td>0</td></tr> <tr><td>1</td></tr> <tr><td>1</td></tr> <tr><td>1</td></tr> <tr><td>1</td></tr> <tr><td>1</td></tr> <tr><td>0</td></tr> <tr><td>0</td></tr> <tr><td>0</td></tr> <tr><td>1</td></tr> </table>	0	0	1	1	1	1	1	0	0	0	1
181	46	male																																									
181	50	male																																									
166	44	female																																									
171	38	female																																									
152	36	female																																									
156	40	female																																									
167	40	female																																									
170	45	male																																									
178	50	male																																									
191	50	male																																									
0																																											
0																																											
1																																											
1																																											
1																																											
1																																											
1																																											
0																																											
0																																											
0																																											
1																																											

8

A simple dataset, with numeric features can simply be represented as a matrix (with a corresponding vector for the labels).

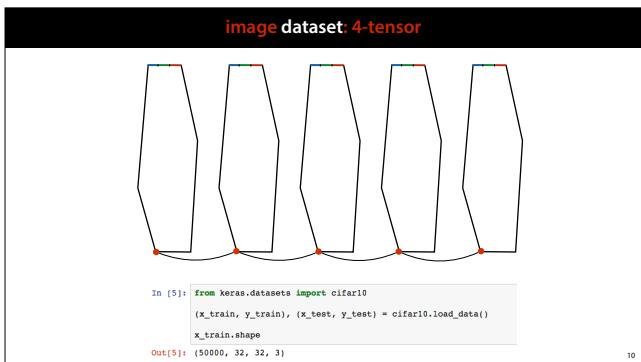
Any categoric features or labels should be converted to numeric features (normally by one-hot coding, as discussed in lecture 4).



But the real benefit in deep learning is not just in representing our standard abstract tasks as features matrices. We want to find ways to represent the raw data, or at least something closer to the raw data, as tensors.

One example is images. These can be represented as 3-tensors. In an RGB image, the color of a single pixel is represented using three values between 0 and 1 (how red it is, how green it is and how blue it is). This means that an RGB image can be thought of as a stack of three matrices, each representing one of the color channels as a grayscale image. This stack is a 3-tensor.

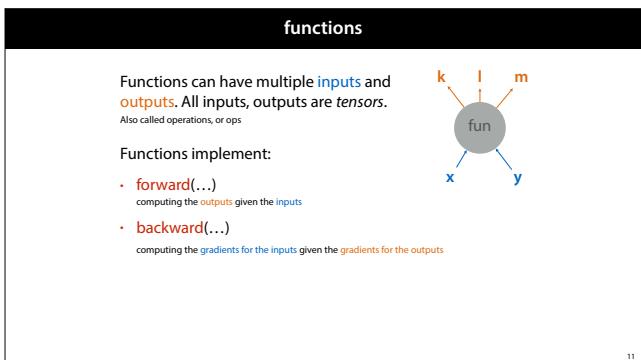
source: <http://www.adsell.com/scanning101.html>



If we then have a *dataset* of multiple images, we get a 4-tensor. The image dimensions we saw already, and one extra, indexing the different images in the data.

This snippet of Keras code shows how this looks in python if we load the CIFAR 10 dataset. The training data contains 50 000 images, each 32 pixels high and 32 pixels wide and with 3 color channels.

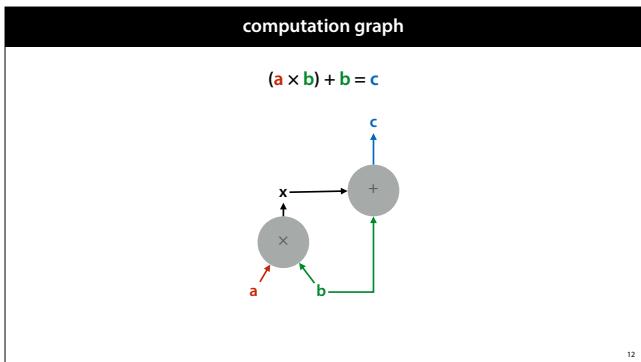
F



The next ingredient we need is **functions** that consume tensors, and spit out new tensors. A function can have multiple inputs and multiple outputs, and all of them are tensors. Non-tensor inputs are sometimes allowed, but when we start computing gradients, we'll only get gradients over the tensor inputs.

Functions exist in many programming environments. All we usually have to do to specify a function is to define how to compute the output given the inputs. In deep learning parlance this is called the **forward** of the function. For a deep learning function, we need to specify one additional thing: a **backward** function. The backward receives the gradients for the **outputs** and computes the gradients for the **inputs**. We'll see some examples of this in the second video.

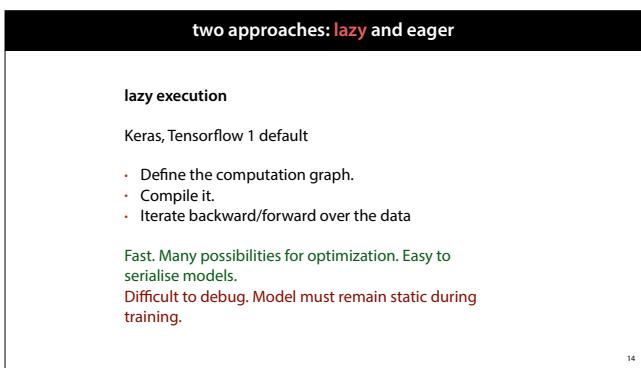
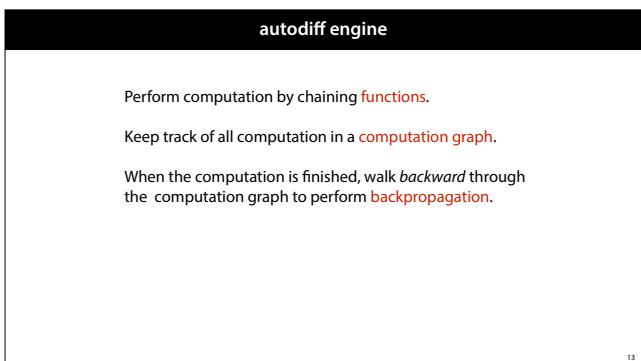
Putting these two ingredients, tensors and function together, we can build a model. We define some functions and we then chain them together into a **computation graph**.



Here is a simple example of a **computation graph**, a directed acyclic graph that shows the data (scalars **a**, **b**, **x**) flowing through the functions.

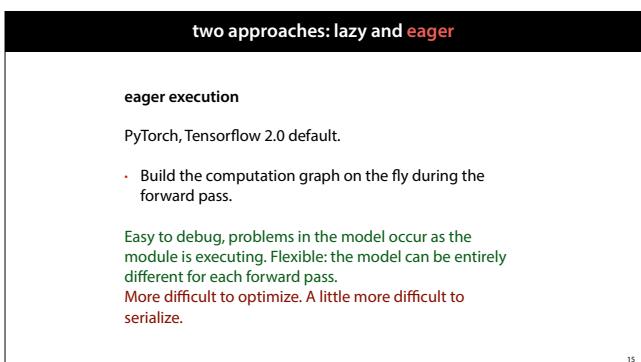
A deep learning system uses a computation graph to execute a given computation (**the forward pass**) and then uses the backpropagation algorithm to compute the gradients for the data nodes with respect to the output (**the backward pass**).

If we are not interested in the specifics of the function being applied, we will omit the circle (as we did in the previous lecture).



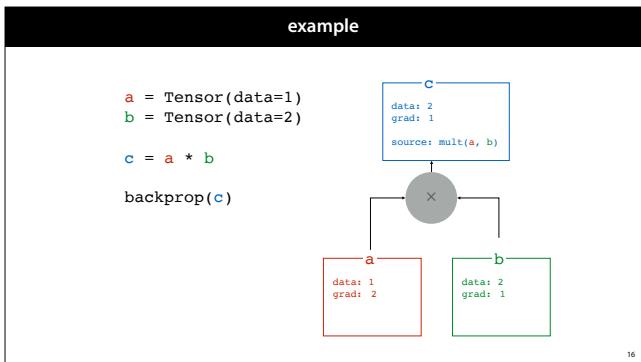
There are two ways of doing this. The first is to use **lazy execution**: you define your computation graph, but you don't place any data in it (only nodes that will hold the data later). Then you compile the computation graph and start feeding data through it.

The drawback of this sort of model is that when something goes wrong during the forward pass, it's very difficult to trace the program error (which happens after you compiled the computation graph) back to where you actually made the mistake (somewhere during definition of the computation graph).



**Eager execution** does not require this kind to predefining of the computation graph. You simply use programming statements to compute the forward pass, for instance multiplying two matrices. The deep learning system then ensures that your matrices are special objects, that keep track of the whole computation, so that when it comes time to do the backward pass, we know how to go back through the computation we performed.

Since eager execution seems to be fast becoming the default approach, we will focus on that, and describe in detail how an eager execution deep learning system works.

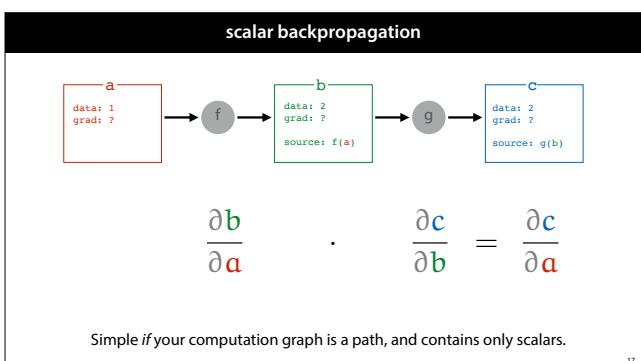


In eager mode deep learning systems, we create a node in our computation graph (called a variable here) by specifying what data it should contain. The result is a tensor object that stores *both* the data, and the gradient over that data (which will be filled later).

Here we create the variables **a** and **b**. If we now apply an operation (a module) to these, for instance to multiply their values, the result has another variable **c**. We compute the data stored in **c** by running the computation, but we also store references to the variables that were used to create **c**, and the module that created it.

Using this graph, we can perform the back propagation from a given starting node. We work our way down the graph computing the derivative of each variable with respect to **c**.

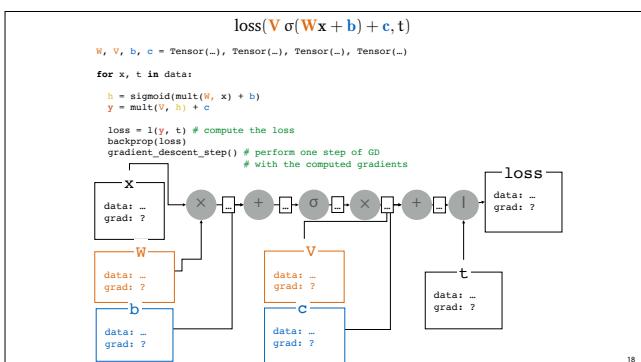
These names and this syntax are loosely inspired by those of PyTorch, but the concepts are similar for all deep learning frameworks.



With a computation graph like this, *if all the data are scalars*, it's very easy to implement backpropagation. Say we're interested in the derivative of **c** over **a**. The chain rule tells us this is the derivative of **b** over **a** times that of **c** over **b**. **c** over **b** is the local derivative for module **g**, and **b** over **a** is the local derivative for module **f**.

Starting at the output we can walk backward, and multiply all the local derivatives we encounter. At each step, the current value is the derivative of the output over the current node.

Next video, we'll see what we need to do when the computation graph has a more complicated structure,

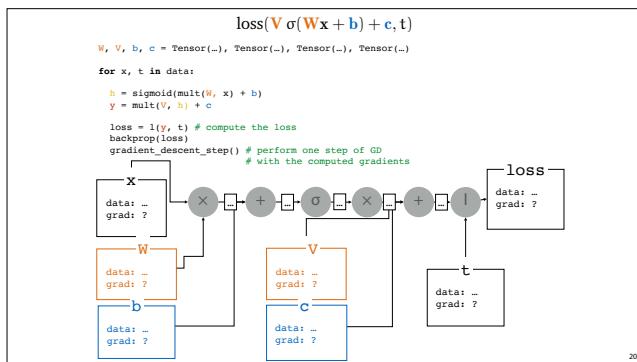
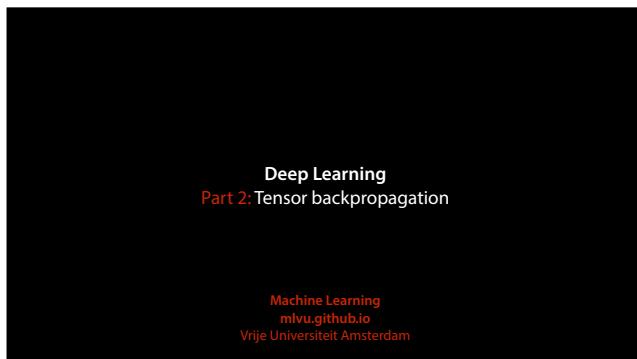


Here's what a training loop would look like for a simple two-layer feedforward network. The computation graph shown below is rebuilt from scratch for every iteration of the training loop, and cleared at the end. The variables **W**, **V**, **b** and **c**, that define our neural network contain tensor data that is saved between iterations, and updated at every step of gradient descent.

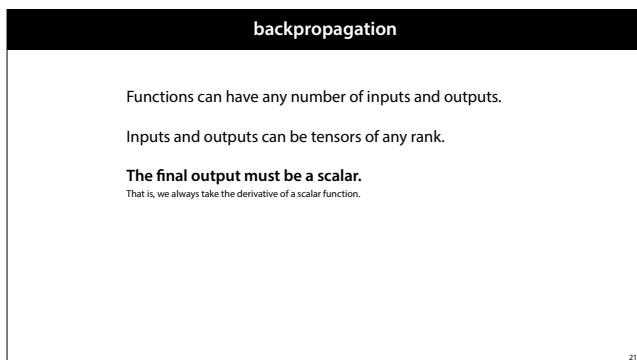
Note that the output of every module is also a variable, with its own data and its own gradient. Note that the multiplications are now matrix multiplications

Once we have our computation graph in place, we have everything we need to start the backpropagation

algorithm. To translate what we learned in the last video to this setting we'll need some extra insights. We'll go over those in the next video.



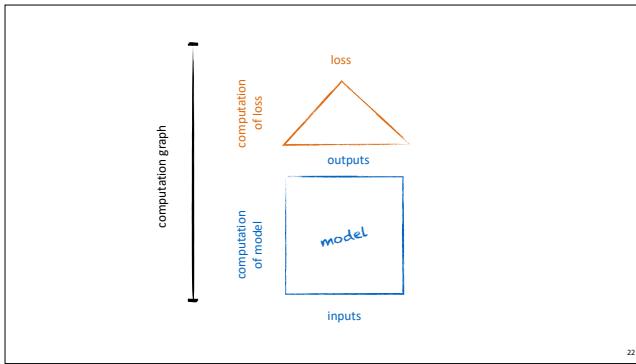
In the previous video we explained how deep learning systems like Pytorch and Tensorflow allow us to build up a computation graph in code. Once we have this computation graph, we can use it to implement backpropagation.



To do so, we will assume these three basic rules.

The last one is essential: the function for which we compute the derivative (with respect to all values in the comp graph) **must** be a scalar. As we shall see, this will allow us to work backwards down the graph, computing the gradients.

Note that this does not prevent us from learning networks with multiple high-dimensional outputs. We just need to define a **single scalar loss** on that output.



Note that this doesn't mean we can only ever train neural networks with a single scalar output. That would be quite boring. Even the multiclass classification model from the previous lecture had three outputs already, and later we want to start building neural networks that generate faces and play chess. All of that is possible: our model can have any number of outputs of any shape and size.

However, the *loss* we define over those outputs needs to map them all to a single scalar value.

The computation graph is always the model, plus the computation of the loss. This way, no matter how complex our model becomes, the computation we're using for backpropagation always has a single scalar output.

## backpropagation revisited

The multivariate chain rule  
Dealing with multiple paths in the comp. graph

Backpropagation with tensors  
Matrix calculus

23

In order to make backpropagation flexible and robust enough to work in this setting, we need to discuss two features that we haven't mentioned yet: how to perform backpropagation if the result depends on the dependent variable along different computation paths, and how to take derivatives when the variables aren't scalars.

## chain rule for multiple inputs

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial a} \frac{\partial a}{\partial x} \quad \frac{\partial c}{\partial y} = \frac{\partial c}{\partial b} \frac{\partial b}{\partial y}$$

24

So far, we've only looked at applying the chain rule to computation graphs that look like paths: a single sequence of function with the output of the last being the input to the next.

If a function has *multiple* inputs, there isn't usually a problem applying the chain rule. If we want the derivative with respect to *x*, we apply the chain rule over the path from *x* to *c*. for this derivative, *b* is a constant, so we can ignore that path in the computation graph.

If we want the derivative with respect to *y*, we apply the chain rule along the other path, taking *a* as a constant.

**iamonds**

$$\frac{\partial c}{\partial x} = ? \frac{\partial c}{\partial a} \frac{\partial a}{\partial x}$$

$$\frac{\partial c}{\partial x} = ? \frac{\partial c}{\partial b} \frac{\partial b}{\partial x}$$

25

We only run into a problem if **c** has two inputs, both depending on **x**.

How do we apply the chain rule here? Over **a** or over **b**?

**multivariate chain rule**

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial a} \frac{\partial a}{\partial x} + \frac{\partial c}{\partial b} \frac{\partial b}{\partial x}$$

26

For such cases, we need the **multivariate chain rule**. It's very simple: to work out the derivative of a function with multiple inputs we just take a single derivative for each input (treating the others as constants), and sum them.

$$\frac{\partial c}{\partial x} = \frac{\partial ab}{\partial x} = \frac{\partial ab}{\partial a} \frac{\partial a}{\partial x} + \frac{\partial ab}{\partial b} \frac{\partial b}{\partial x}$$

$$= b \frac{\partial a}{\partial x} + a \frac{\partial b}{\partial x}$$

27

The multivariate chain rule can be used to derive many rules for derivatives you should already know. For instance, if we make **c** the product of **a** and **b**, applying the multivariate chain rule gives us the product rule.

$$\frac{\partial c}{\partial x} = \sum_i \frac{\partial c}{\partial a_i} \frac{\partial a_i}{\partial x}$$

28

If **c** has more than two inputs, the multivariate chain tells us to sum over all of them.

## backpropagation revisited

**The multivariate chain rule**  
Dealing with multiple paths in the comp. graph

Backpropagation with tensors  
Matrix calculus

With that, we know how to apply the chain rule to any kind of computation graph.

Next, we need to figure out how to make backpropagation work in settings where our inputs and outputs are *tensors*.

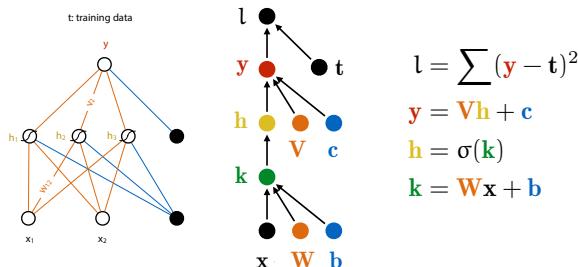
## backpropagation revisited

**The multivariate chain rule**  
Dealing with multiple paths in the comp. graph

Backpropagation with tensors  
Matrix calculus

In order to make backpropagation flexible and robust enough to work in this setting, we need to discuss two features that we haven't mentioned yet: how to perform backpropagation if the result depends on the dependent variable along different computation paths, and how to take derivatives when the variables aren't scalars.

## break up your computation into modules



The first step of applying backpropagation is to break our computation into modules. For our feedforward neural network, this is a natural way to draw the computation graph. Note that both the model parameters and the inputs are tensor nodes in the computation graph.

## matrix calculus

Can we express the local derivatives in terms of the *tensor* inputs and outputs?

$$l = \sum (\mathbf{y} - \mathbf{t})^2$$

$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{c}$$

$$\mathbf{h} = \sigma(\mathbf{k})$$

$$\mathbf{k} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

The next step is to work out the local derivatives. We would like to have a chain rule like the one shown on the right, and then to work out how to compute those local derivatives efficiently.

What does it mean to take the derivative of a vector, or with respect to a matrix?

## matrix calculus

Can we express the local derivatives in terms of the *tensor* inputs and outputs?

$$l = \sum (\mathbf{y} - \mathbf{t})^2$$

$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{c}$$

$$\mathbf{h} = \sigma(\mathbf{k})$$

$$\mathbf{k} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\frac{\partial l}{\partial \mathbf{W}} = \frac{\partial l}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}$$

The next step is to work out the local derivatives. We would like to have a chain rule like the one shown on the right, and then to work out how to compute those local derivatives efficiently.

What does it mean to take the derivative *of a vector, or with respect to a matrix?*

function returns a			
	scalar	vector	matrix
scalar	scalar	vector	matrix
vector	vector	matrix	?
matrix	matrix	?	?

Vector-to-scalar function  $f(\mathbf{x}) = y$   
n inputs, 1 output, n possible scalar derivatives.  
*Vector of derivatives. (gradient)*

Scalar-to-vector function  $f(\mathbf{x}) = \mathbf{y}$   
n inputs, 1 output, n possible scalar derivatives.  
*Vector of derivatives.*

Vector-to-vector function  $f(\mathbf{x}) = \mathbf{y}$   
n inputs, m output,  $n \times m$  possible scalar derivatives. *Matrix of derivatives.*

Matrix-to-vector function  $f(\mathbf{X}) = \mathbf{y}$   
 $n \times m$  inputs, k output,  $n \times m \times k$  possible scalar derivatives. *3-tensor of derivatives?*

$$\frac{\partial l}{\partial \mathbf{W}} = \frac{\partial l}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}$$

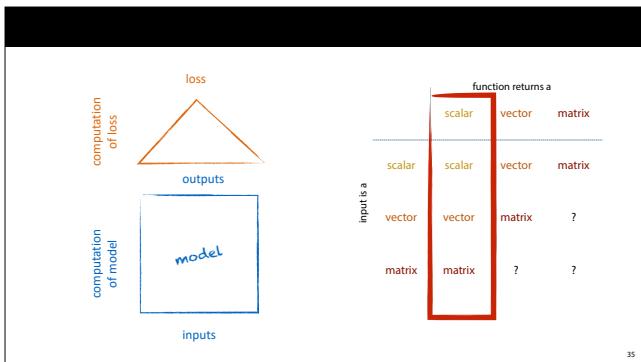
There are ways to define the derivative of a function with respect to a matrix or a vector. In general, if we have a function with a tensor input and a tensor output, we can take a large number of scalar derivatives by taking the derivative of one of its outputs with respect to one of its inputs. The gradient is an example of this: we have a function from a vector to a scalar, so we take all scalar derivatives of the output with respect to one of the inputs, and collect them into a vector.

If we have a function from a vector to a scalar, we can collect all derivatives of one output with respect to the single input. This can also be neatly represented in a vector.

If we have a vector-to-vector function, we can take all scalar derivatives of one of the m outputs with respect to one of the n inputs. This is best represented by a n-by-m *matrix*.

If we go higher, like a matrix-to-vector function, we get so many derivatives that we need a 3-tensor to represent them. And this is where we run into trouble.

For matrices and vectors, multiplication is still defined, and works similarly to scalar multiplication. That means that so long as our derivatives are only matrices and vectors, we can still hope for a functional chain rule, where we can work out the local derivatives compute them and multiply them together. But this already breaks down in the case of the feedforward network. If we imagine what a chain of local derivatives for our computation graph might look like, we'd get something like this expression at the bottom. even for something so simple as a feedforward network, one of the factors is already the derivative of a vector over a matrix. This means the result should be represented in a 3-tensor, for which multiplication isn't defined (or at least not unambiguously), so that we can never multiply all the local derivatives to give us the global derivatives.



Our saving grace is the fact that we assume our function as a whole always has a **scalar output**. This means that whatever we are doing, the only derivatives we ever want to end up with in the end are those of the loss (a scalar) with respect to some tensor in our computation graph. This means that we can stay in the leftmost column of this matrix.

### the gradient

We will call  $\frac{\partial l}{\partial \mathbf{W}}$  the gradient of  $l$  with respect to  $\mathbf{W}$ . Or the gradient *for*  $\mathbf{W}$ .

Commonly written as  $\nabla_{\mathbf{W}} l$

Nonstandard convention:  $\nabla_{\mathbf{W}} l$  has the *same shape* as  $\mathbf{W}$ .

$$[\nabla_{\mathbf{W}} l]_{ijk} = \frac{\partial l}{\partial W_{ijk}}$$

The only derivatives we will ever be interested in, ultimately, are the derivatives of the loss with respect to one of the inputs of the computation graph (the inputs to the network, or the parameters). For these, we can always represent the the collection of all derivatives, by giving it the same shape as the tensor we're taking the derivative over.

In the example shown,  $\mathbf{W}$  is a 3-tensor. The gradient of  $l$  wrt  $\mathbf{W}$  has the same shape as  $\mathbf{W}$ , and at element  $(i, j, k)$  it holds the scalar derivative of  $l$  wrt  $\mathbf{W}_{ijk}$ .

With these rules, we can use tensors of any shape and dimension and always have a well defined gradient.

Note that in other fields, the gradient often has a *different shape* from the thing we're taking the gradient for. If the function takes column vectors, the gradient is defined as a row vector. That's because the gradient is usedas an *operator*, defining a function on the original vector space. In our case, we are not interested in using the gradient in this way: it only ever defined a direction in our model space, which will help us search, so for us it makes more sense to have the gradient be the same shape and size as the points in the model space.

### new notation

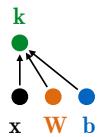
$$\nabla_{\mathbf{A}} l = \mathbf{A}^{\nabla} \quad A_{ij}^{\nabla} = \frac{\partial l}{\partial A_{ij}}$$

$$\nabla_{\mathbf{b}} l = \mathbf{b}^{\nabla}$$

This nabla notation

### for example

$$\mathbf{k} = \mathbf{W}\mathbf{x} + \mathbf{b}$$



`forward( $\mathbf{W}, \mathbf{x}, \mathbf{b}$ ): compute  $\mathbf{W}\mathbf{x} + \mathbf{b}$`

`backward( $\mathbf{k}^\nabla$ ): compute  $\mathbf{W}^\nabla, \mathbf{x}^\nabla, \mathbf{b}^\nabla$`

38

With these principles in place, we can apply backpropagation in a tensor friendly way. Instead of computing the local derivatives first, and then multiplying to compute the global derivatives, we accumulate the product of the local derivatives directly.

This is the first layer of our feedforward network. It has three inputs  $\mathbf{W}$ ,  $\mathbf{x}$ , and  $\mathbf{b}$ , and one output  $\mathbf{k}$ . As we saw earlier, the local derivative, consists of a 3-tensor of scalar derivatives.

The **forward function** computes the unactivated values of the first layer, given the inputs  $\mathbf{x}$ , weights  $\mathbf{W}$  and bias  $\mathbf{b}$ .

The **backward function** is given the derivative of the loss for  $\mathbf{k}$ , and should output the derivatives of the loss for  $\mathbf{W}$ ,  $\mathbf{x}$ , and  $\mathbf{b}$ .

We can't

We'll look at the first as an example. The local derivative  $\mathbf{k}$  over  $\mathbf{W}$  is a 3 tensor which is difficult to work with, and would take up a large amount of memory, so we don't want to compute it explicitly. Let's see if we can work out a more efficient expression for the required product.

### working out derivatives for high rank tensors

1. Describe the problem in terms of *scalar derivatives*.
2. Apply the scalar (multivariate) chain rule.
3. Rewrite the scalar computations as tensor operations.

To work out what such a backward function should return, we can follow this plan.

39

A diagram showing a computation graph for a linear layer and its derivative calculation. The graph starts with input  $W_{32}$ , which is multiplied by  $x_1$  and  $x_2$  to produce  $k_1$ . This is then added to  $b_1$  to produce  $l$ . The graph is represented as follows:

```

graph LR
    W32[W32] --> prod1[W]
    prod1 --> k1((k1))
    prod1 --> prod2[W]
    prod2 --> k2((k2))
    prod2 --> prod3[W]
    prod3 --> k3((k3))
    k1 --> sum1[+]
    b1((b1)) --> sum1
    sum1 --> l((l))
  
```

Below the graph, the derivative calculation is shown:

$$\begin{aligned}
 \frac{\partial l}{\partial W_{32}} &= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial k_i}{\partial W_{32}} \\
 &= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial (\mathbf{W}\mathbf{x} + \mathbf{b})_i}{\partial W_{32}} \\
 &= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial (\mathbf{W}_i \mathbf{x} + \mathbf{b}_i)}{\partial W_{32}} \\
 &= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial \sum_j (\mathbf{W}_{ij} x_j)}{\partial W_{32}} \\
 &= \sum_{ij} \frac{\partial l}{\partial k_i} \frac{\partial \mathbf{W}_{ij} x_j}{\partial W_{32}} = \frac{\partial l}{\partial k_3} x_2
 \end{aligned}$$

40

To work out a scalar derivative we pick an arbitrary element of  $\mathbf{W}$ , say  $W_{32}$ , and work out the derivative for that. Note that since we are using matrix notation  $W_{32}$  is the weight from input 2 to output 3. In the previous videos the subscripts were the other way around.

If we think of this as a computation graph with scalar nodes,  $\mathbf{k}$  just represents different inputs to the function that ultimately computes  $l$ . That means that the derivative of  $l$  over  $W_{32}$  is just the sum of the derivatives through each element of  $\mathbf{k}$ . This works whatever the rank and shape of  $\mathbf{k}$ ; it could be a huge 9-tensor, and all we have to do is flatten it, and sum over its derivatives. Note that these are the derivatives that we are given.

At the end, we see that the scalar derivative we're interested in is the second element of the vector that we are given, times the third element of the input  $\mathbf{x}$ .

**vectorize**

$$\mathbf{W}_{32}^\nabla = \frac{\partial l}{\partial \mathbf{W}_{32}} = \frac{\partial l}{\partial k_3} x_2 = k_3^\nabla x_2$$

$$\mathbf{W}^\nabla = \begin{bmatrix} k_1^\nabla x_1 & k_1^\nabla x_2 \\ k_2^\nabla x_1 & k_2^\nabla x_2 \\ k_3^\nabla x_1 & k_3^\nabla x_2 \end{bmatrix} = \mathbf{k}^\nabla \mathbf{x}^T$$

41

We don't actually want to compute the scalar derivatives one by one like this, but at least now we know what expected of us. We can write down what all the elements of the matrix  $\mathbf{W}^\nabla$  look like, and see if we can find some clever way to figure out how to compute this matrix using simple linear algebra operations, instead of filling the elements of the matrix one by one. This is called **vectorizing**: expressing an algorithm in single matrix operations rather than a series of loops.

In this case, we can note that is simple the outer product of the vector  $\mathbf{k}^\nabla$  which we were given and the input vector  $\mathbf{x}$ . Multiplying these two will give us all the derivatives we're interested in, in a single operation.

**for example**

$$\mathbf{k} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

forward( $\mathbf{W}, \mathbf{x}, \mathbf{b}$ ): compute  $\mathbf{W}\mathbf{x} + \mathbf{b}$

backward( $\mathbf{k}^\nabla$ ): compute  $\mathbf{W}^\nabla = \mathbf{k}^\nabla \mathbf{x}^T$   
 $\mathbf{x}^\nabla = \mathbf{W}^T \mathbf{k}^\nabla$   
 $\mathbf{b}^\nabla = \mathbf{k}^\nabla$

42

The gradients for  $\mathbf{x}$  and  $\mathbf{b}$  can be worked out in the same way.

**backpropagation**

43

With these ingredients, the rest of backpropagation is a breeze. We compute a forward pass, remembering the intermediates and then we walk **backwards** down the graph. At each module we call its `backward()` function with the gradients for its outputs and receive the gradients for its inputs. So long as we do this in the right order, we can be sure that we will compute all gradients for all nodes in the graph.

## in summary

**model:** the application of functions to tensors

each function defines a `backward()` function:  
given the gradient over the **outputs**, compute the  
gradient over the **inputs**.

backpropagation walks backwards through the graph,  
accumulating the gradient product.

44

## working out the backward function

Usually not necessary, only if you write your own module  
But it's good to understand the principle

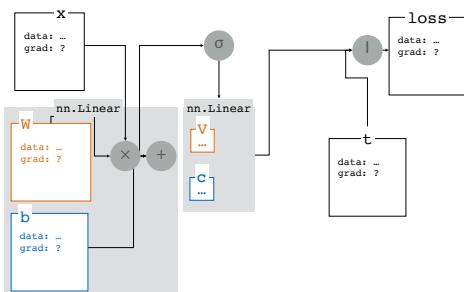
- Phrase the problem in terms of scalar derivatives: work out the derivative for one element of the input.
- Use the multivariate chain rule: sum over all elements of the output variable.
- Work out the general solution in terms of matrix operations

Further reading:  
<https://dvlu.github.io>  
<https://compsci682-fa18.github.io/docs/vecDerivs.pdf>  
<http://cs231n.stanford.edu/handouts/derivatives.pdf>

45

Working out a backward function is not usually necessary in practice: deep learning frameworks provide a large number of pre-built modules that you can chain together to do almost anything. Only when you write your own module, do you need to implement the backward and forward functions.

## modules / layers



46

Most deep learning frameworks also have a way of combining model parameters and computation into a single unit, often called a **module** or a **layer**.

In this case a Linear module (as it is called in Pytorch) takes care of implementing the computation of a single layer of a neural network (sans activation) and of remembering the weights and the bias. These modules combine existing functions together with tensors. Implementing a module is easy, you only define the forward part of the computation. The backward is done automatically, because everything is defined in terms of functions that already have a backward implemented.

## backpropagation revisited

~~The multivariate chain rule~~

Dealing with multiple paths in the comp. graph

~~Backpropagation with tensors~~

Matrix calculus

In order to make backpropagation flexible and robust enough to work in this setting, we need to discuss two features that we haven't mentioned yet: how to perform backpropagation if the result depends on the dependent variable along different computation paths, and how to take derivatives when the variables aren't scalars.

47

<https://github.com/dlvu/vugrad>

```
Raw Blame
```

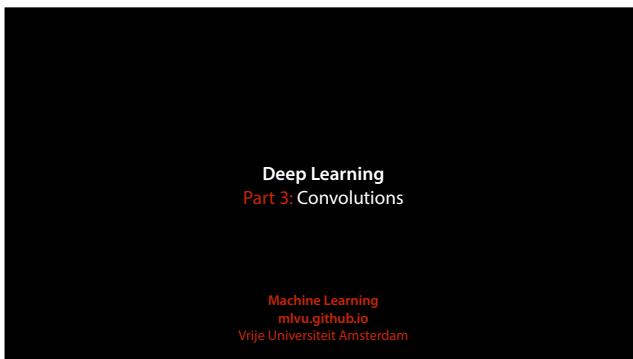
```
338 lines | 245 sloc | 18.8 KB

1 import numpy as np
2
3 ...
4 This module contains the core components of the autodiff system: the two types of nodes that make up the computation graph
5 (TensorNodes and Opnodes) and the class definition of an Op.
6
7 The main algorithm of backpropagation is implemented recursively in the backward functions of the nodes.
8 ...
9
10
11 class TensorNode:
12     ...
13     Represents a value node in the computation graph: a tensor value linked to its
14     computational history
15     ...
16
17     def __init__(self, value: np.ndarray, source=None):
18         ...
19
20         #Baran value: A numpy array,
21         #param source: The opnode that created this tensor node. Leave this blank if you create a TensorNode manually.
22
23         self.value = value # the raw value that this node holds
24         self.source = source # the opnode that produced this Node
25
26         self.grad_fn = np.zeros_like(value.shape) # this is where we will put the gradient when we compute the loss
27
28         # the gradient is defined as a tensor with the same dimensions as the value. At element 'index' it contains
```

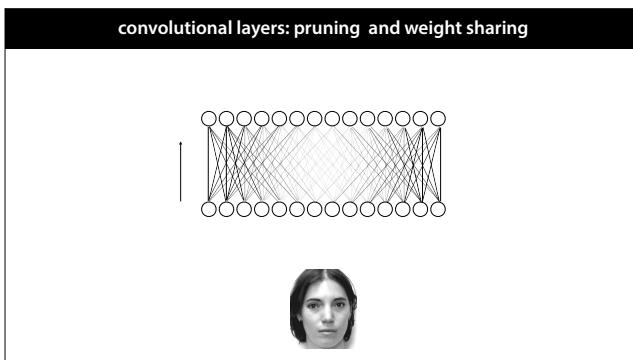
And with that, we have all the ingredients for a modern deep learning framework.

If you'd like to see what this looks like in practice, click this link to see a very minimal implementation of such a deep learning system, in about 300 lines of code. If you'd like to get your hands dirty and start training neural networks, check out the fourth and fifth worksheets.

In the next videos, we see what we can build in systems like this besides simple feedforward networks. Specifically, we'll look at convolutional neural networks.

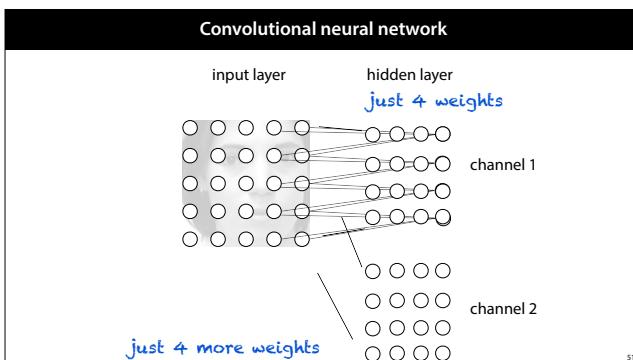


Deep Learning  
Part 3: Convolutions



To get started with deep learning, let's look at our first special layer. That is, a layer that is now just a fully connected linear transformation of a vector, but a layer whose shape is determined by some knowledge about its purpose. In the case of the convolution, its purpose is to consume *images*.

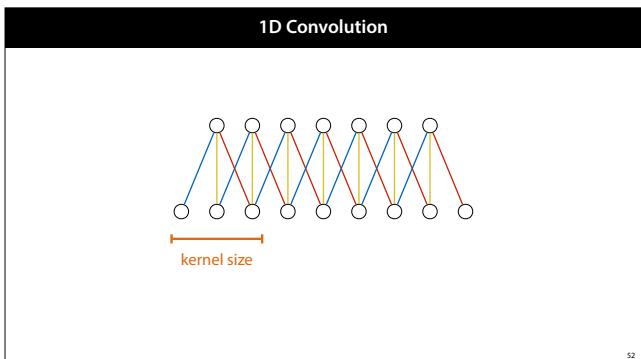
We know that images form a grid, and we can use this information to get far fewer connections, and far fewer weights in the layer than a fully connected layer.



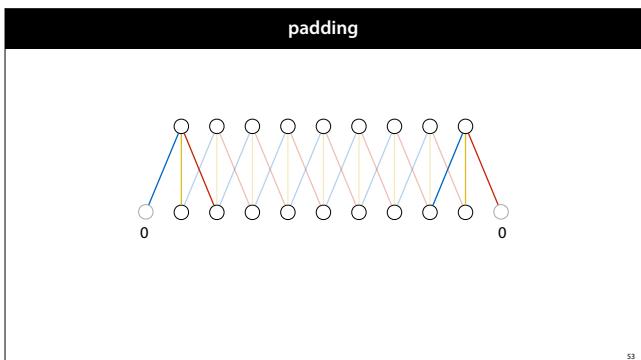
We connect each node in the hidden layer just to a small  $n$  by  $n$  neighbourhood in the input (here  $n=2$ ); there are no connections to any other pixels. We do this for each such  $n \times n$  neighbourhood in the input. For an input image of 5 by 5 pixels, this gives us an input layer of 25 nodes, and a hidden layer of 16 nodes (which we've also arranged in a grid). Each node in the hidden layer has just 4 incoming connections. What's more, we set the 4 weights of these incoming connections to be the same for each of the 16 nodes in the hidden layer.

To extend the hidden layer, we can add additional **channels** to the hidden layer. For an extra channel we follow the same procedure but with 4 new weights. If, as shown here, we have a 5 by 5 input layer with 4

pixel neighbourhoods, and two maps, we get a network with 25 inputs and 32 nodes in the hidden layer. In a traditional feedforward network, that would give us  $25 \times 32$  connections with as many weights. Here, we have just  $32 \times 4$  connections, and only 8 weights.



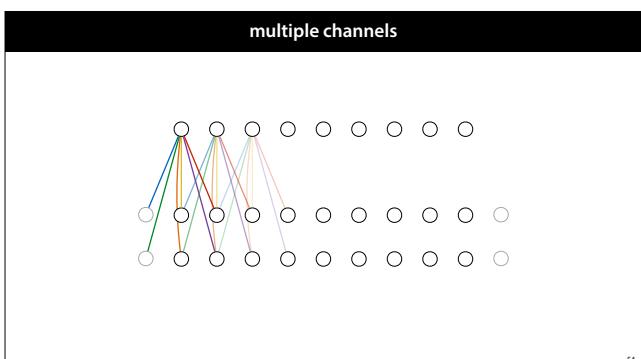
Here is how it looks if the input is 1D (a sequence of units rather than a grid). Note that the connection colors indicate shared weights (that is, every blue connection has the same weight).



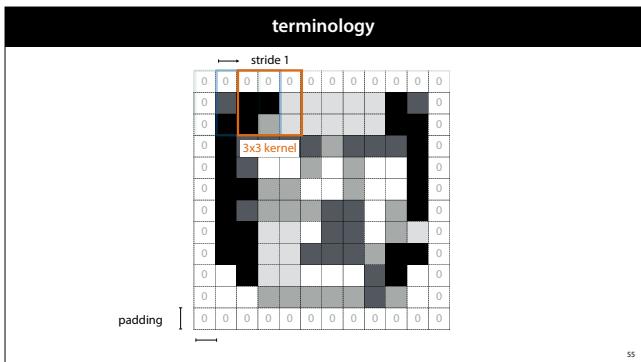
The drawback with the previous picture is that the inputs on the side contribute to only one hidden unit, and the one next to it to only two

To combat this, we can add *padding*: extra units, usually with a fixed value set to zero. Because of this padding, the number of output is the same as the number of inputs, and the actual units on the side contribute to more nodes.

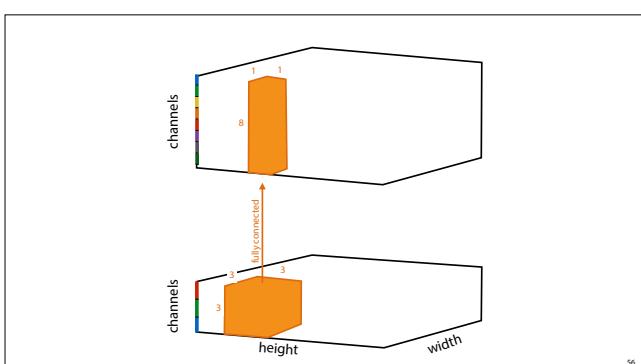
To achieve the same number of units in the output as in the input (before padding), we must set the number of units padded on both sides to  $\text{floor}(\text{kernel\_size}/2)$ . This is sometimes called “same padding”.



If our input has multiple channels (like one color channel for each pixel), the standard approach is to add new weights for the new channel. Note that these are repeated along the spatial dimension(s) just like the other weights. The same approach is used to create multiple output channels.

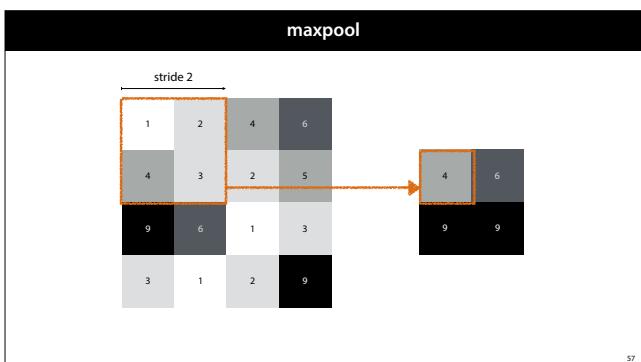


Here is the view in 2 dimensions. We normally slide the kernel one step each time (this is called a stride of 1), but we can also increase the stride to lower the output resolution.



Used in this way, the convolution layer transforms the input, a 3-tensor, into another 3-tensor with the same resolution and more channels.

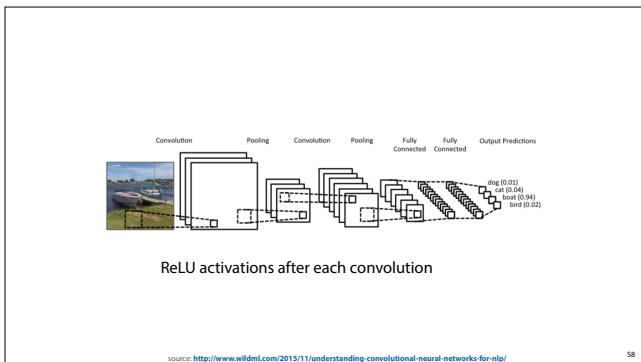
Between the two orange boxes, everything is fully connected (every channel of every pixel in the lower boxer is connected by unique weight to every channel of every pixel in the top box).



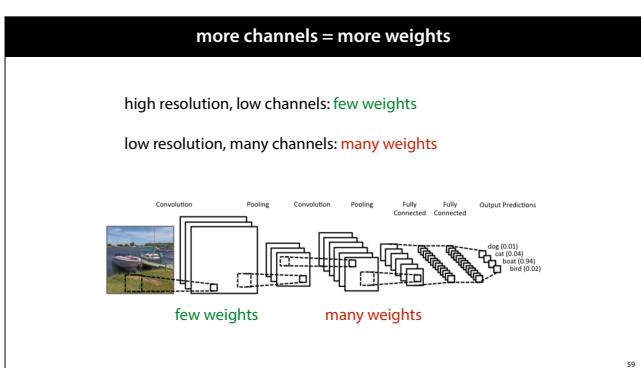
We chain these convolutions together, but after a while (as the number of channels grows) we'd like the resolution to decrease so we're gradually looking at less specific parts of the image, but we have more information (more channels) about that part of the image.

The max pooling layer does this for us, it divides the image into n-by-n squares, and returns the maximum value from each square. Average pooling (returning the average over each square) is also possible, but max pooling is usually more effective.

Most likely, this is because during the backward pass, the whole gradient flows back down one of the pixels, instead of dividing over all four. This gives the backpropagation a sparse character: it focuses only on the most important paths in the computation graph instead of dividing its attention over all of them.



We extend this principle for many more layers. At each step the maps of the layers get smaller, and we add more maps. Eventually, we add one or two fully connected layers, and a (softmax) output layer (if we're doing image classification).



### worksheet 4

```

from keras.models import Sequential
from keras.layers import Dense, Activation, Conv2D, MaxPool2D, Dropout, Flatten

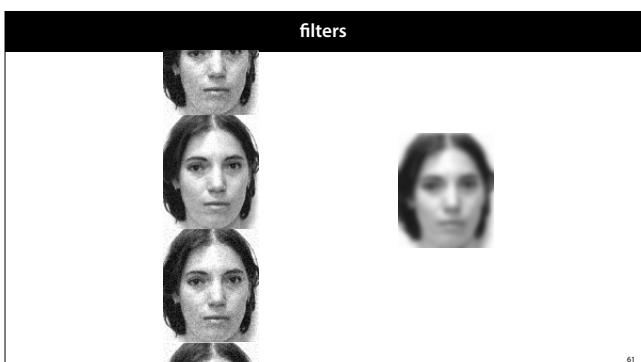
model = Sequential()
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25)) # Dropout 25% of the nodes of the previous layer during training
model.add(Flatten()) # Flatten, and add a fully connected layer
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax')) # Last layer: 10 class nodes, with dropout
model.summary()

from keras.optimizers import Adam
optimizer = Adam()
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model, iterating on the data in batches of 32 samples
model.fit(x_train, y_train, epochs=15, batch_size=32, validation_split=1/6)

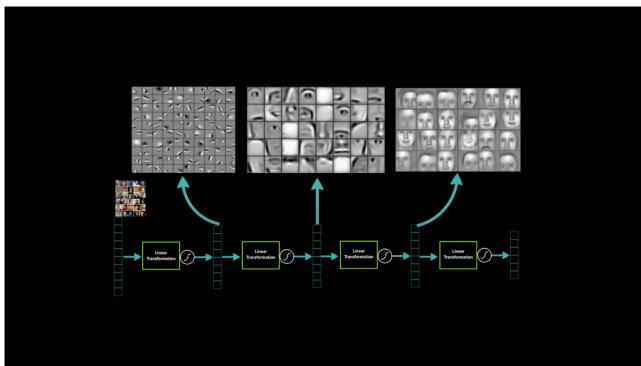
```

60



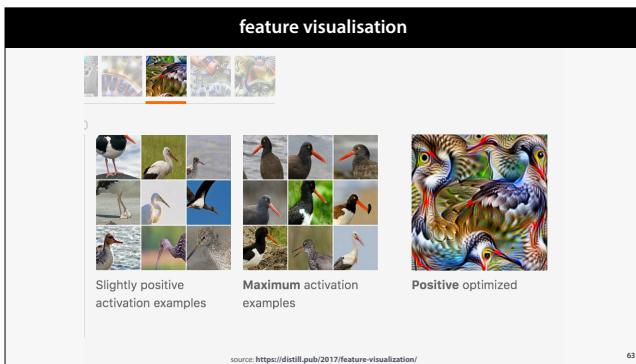
Each first layer gives us several **convolution filters** (one per channel). Here is an example: the Gaussian convolution. It takes a pixel neighbourhood and averages the pixels in it, creating a blurred result of the input (this is just one transformation that a convolution filter can perform, depending on the weights, many other operations are possible).

While Gaussian blur, may seem to be throwing away valuable information, what we actually get is a representation that is *invariant* to noise. All these noisy input images in the left will be mapped to the same image on the right. We can do the same thing to create representations invariant to, for instance, small translation.



Here are the results of a real convolutional network trained to detect faces. The small grayscale images shows a typical image that each node in one of the layers responds to. Those for the first layers can be thought of as edge detectors: if there is a strong edge in a particular part of the image, the node lights up. The second combine these into detectors for parts of images: eyes noses, mouths, etc. The third combine these into detectors for complete faces.

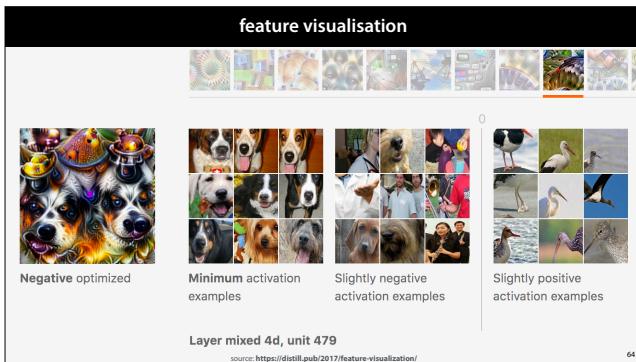
source: <https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/>



Here is a feature visualisation example for a more recent network trained on Imagenet, a collection of 14 million images with diverse subjects.

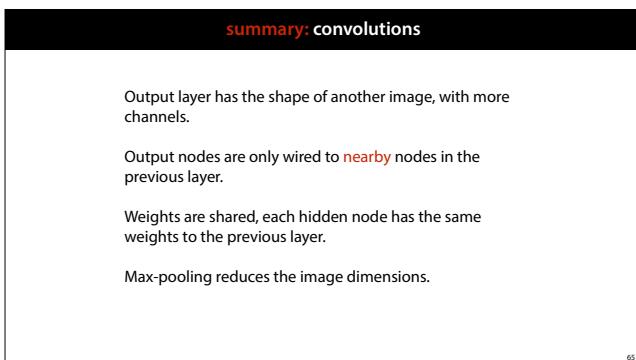
To find the image on the right, the authors took one node high up in the network, and instead of optimising the weights to minimise the loss, they optimised the input to maximise the activation of that node.

They also searched the dataset for natural images that caused a high activation in that particular node.



The opposite is also possible: searching for an input that cause minimal activation.

<https://distill.pub/2017/feature-visualization/>



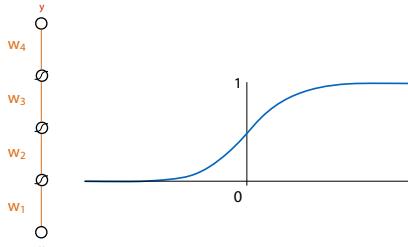
## making it work

How to make *deep* neural nets work.

- Overcoming vanishing gradients  
Proper initialisation, ReLUs over sigmoids
- Minibatch gradient descent
- Optimizers  
(Nesterov) momentum, Adam
- Regularizers  
L1, L2, Dropout

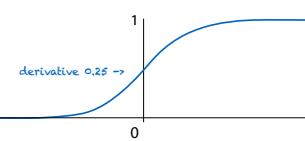
These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

## vanishing gradient problem



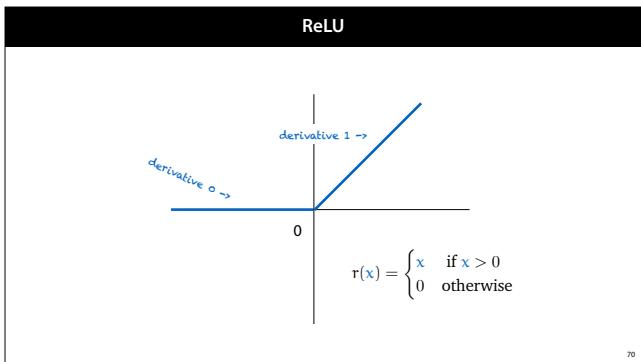
Here is a simple network to illustrate the problem of **vanishing gradients**. The question is how should we initialize its weights? If we set them too large, the activations will hit the rightmost part of the sigmoid. Consequently, the local gradient for each node will be very close to zero. That means that the network will never start learning.

If we go the other way, and make the weights large negative numbers, then we hit the leftmost part of the sigmoid and we have the same problem.



Even if the value going in to the sigmoid is close enough to zero, we still end up with a derivative of only one quarter. This means that propagating the gradient down the network, it will still go to zero with many layers.

We could fix this by squeezing the sigmoid, so its derivative is 1, but it turns out there is a better and faster solution that doesn't have any of these problems.



The ReLU activation preserves the derivatives for the nodes whose activations it lets through. It kills it for the nodes that produce a negative value, of course, but so long as your network is properly initialised, about half of the values in your batch will always produce a positive input for the ReLU.

There is still the risk that during training, your network will move to a configuration where a neuron always produces negative input for every instance in your data. If that happens, you end up with a dead neuron: its gradient will always be zero and no weights below that neuron will change anymore (unless they also feed into a non-dead neuron).

### initialization

Make sure your data is standardized/normalized  
i.e. the mean is close to 0, and the variance is close to 1 in every direction.

initialise weights  $\mathbf{W}$ :

- Make  $\mathbf{W}$  a random orthogonal matrix (eigenvalues all 1).
- Glorot uniform:

$$w_{ij} \sim U\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

71

There are two standard initialisation mechanisms. The idea of both is that we assume that the layer input is (roughly) distributed so that its mean is 0 and the variance is 1 in every direction (we must standardise or normalise the data so this is true for the first layer).

The initialisation is then designed to pick a random matrix that keeps these properties true (in a stochastic sense).

### making it work

How to make deep neural nets work.

- Overcoming vanishing gradients  
Proper initialisation, ReLUs over sigmoids
- Minibatch gradient descent
- Optimizers  
(Nesterov) momentum, Adam
- Regularizers  
L1, L2, Dropout

72

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

### minibatch gradient descent

Like stochastic gradient descent, but with small batches of instances, instead of single instances.

Smaller batches: more like stochastic GD, **more noisy, less parallelism**.

Bigger batches: more like regular GD, **more parallelism**, limit is (GPU) memory

General advice, keep it between 16 and 128 instances.

73

## making it work

How to make deep neural nets work.

- Overcoming vanishing gradients

Proper initialisation, ReLUs over sigmoids

- Minibatch gradient descent

- Optimizers

(Nesterov) momentum, Adam

- Regularizers

L1,L2, Dropout

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

74

## optimizers

Attempt to adapt the gradient descent update rule to improve convergence.

- Momentum
- Nesterov momentum
- RMSProp
- AdaGrad
- AdaDelta
- AdaMax
- Adam
- Nadam

General advice, use Adam, try Nesterov momentum if it doesn't work.

75

## momentum

plain gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \text{loss}(\mathbf{w})$$

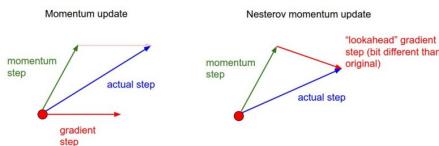
with momentum

$$\begin{aligned}\mathbf{v} &\leftarrow \mu \mathbf{v} - \eta \nabla \text{loss}(\mathbf{w}) \\ \mathbf{w} &\leftarrow \mathbf{w} + \mathbf{v}\end{aligned}$$

If gradient descent is a hiker in a snowstorm, then momentum gradient descent is a boulder rolling down a hill. The gradient doesn't affect its movement directly, it acts as a **force** on a moving object. If the gradient is zero, the updates continue in the same direction as the previous iteration, only slowed down by a "friction constant"  $\mu$ .

76

## Nesterov momentum



source: <http://cs231n.github.io/neural-networks-3/#sgd>  
(recommended reading)

Nesterov momentum is a slight tweak. In regular momentum, the actual step taken is the sum of two vectors, the momentum step (a step in the direction we took last iteration) and a gradient step (a step in the direction of steepest descent at the current point).

Since we know that we are taking the momentum step anyway, we might as well evaluate the gradient *after* the momentum step. This will make the gradient slightly more accurate.

77

## adam

Normalize the gradients: keep a running mean  $\mathbf{m}$  and (uncentered) variance  $\mathbf{v}$  for each parameter over the gradient. Subtract these instead of the gradient.

$$\begin{aligned}\mathbf{m} &\leftarrow \beta_1 * \mathbf{m} + (1 - \beta_1) \nabla \text{loss}(\mathbf{w}) \\ \mathbf{v} &\leftarrow \beta_2 * \mathbf{v} + (1 - \beta_2) (\nabla \text{loss}(\mathbf{w}))^2 \\ \mathbf{w} &\leftarrow \mathbf{w} - \eta \frac{\mathbf{m}}{\sqrt{\mathbf{v}} + \epsilon}\end{aligned}$$

Comes with two extra hyperparameters, but the defaults are usually fine.

Adam combines the idea of momentum with the idea that in large, complex networks each weight should have its own learning rate. Different weights perform very different functions, so ideally we want to look at the properties of the loss landscape for each weight (the sizes of recent gradients) and scale the “global learning rate” by these.

The bigger the recent gradients, the bigger we want the learning rate to be. However, if there is a lot of variance in the recent gradients, we want to reduce the learning rate (because the landscape is unpredictable). Thus, if we scale the learning rate by the mean  $\mathbf{m}$  over the recent gradients, and divide that by the square root of the variance  $\mathbf{v}$  (plus some small epsilon to avoid division by zero), we end up with a direction that uses recent information about the loss landscape to adapt the gradient to the local topology.

$\mathbf{m}$  and  $\mathbf{v}$  are computed as an exponential moving average. This means that the current gradient weights the most, and the influence of recent gradients decays exponentially (but all play *some* part in the total sum).

## making it work

How to make deep neural nets work.

- Overcoming vanishing gradients  
Proper initialisation, ReLUs over sigmoids
- Minibatch gradient descent
- Optimizers  
(Nesterov) momentum, Adam
- Regularizers  
L1, L2, Dropout

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

## regularizers

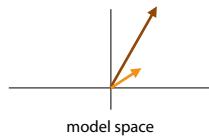
The bigger your model, the greater the capacity for **overfitting**.

Regularizers attempt to pull the model back towards more simple models, without eliminating the more complex solutions.

### L2 regularizer

$$\text{loss}_{\text{reg}} = \text{loss} + \lambda \|\theta\|$$

$$\text{loss}_{\text{reg}} = \text{loss} + \lambda \|\theta\|^2 = \text{loss} + \lambda \theta^T \theta$$



81

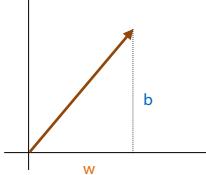
The L2 regularizer considers models with small parameters to be simpler (and therefore preferable). It adds a penalty to the loss for models with larger weights.

To implement the regularizer we simply compute the L2 norm of all the weights of the model (flattened into a vector). We then add this to the loss multiplied by hyper parameter `lambda`. Thus, models with bigger weights get a higher loss, but if it's worth it (the original loss goes down enough), they can still beat the simpler models.

Theta is a vector containing all parameters of the model (it's also possible to regularise only a subset).

### vector norm

$$\theta = \begin{pmatrix} w \\ b \end{pmatrix}$$



$$\|\theta\| = \sqrt{w^2 + b^2}$$

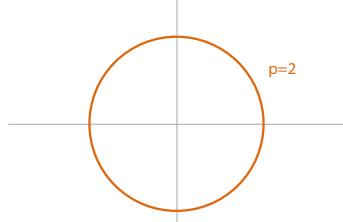
$$\|\theta\|^p = \sqrt[p]{w^p + b^p}$$

82

We can generalise the L2 norm to an  $l_p$  norm by replacing the squares with some other number  $p$ .

### $l_p$ norm

$$\|\theta\|^p = \sqrt[p]{w^p + b^p}$$

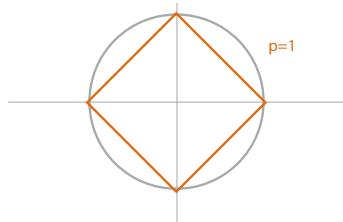


83

For the L2 norm, the set of all points that have distance 1 to the origin form a circle.

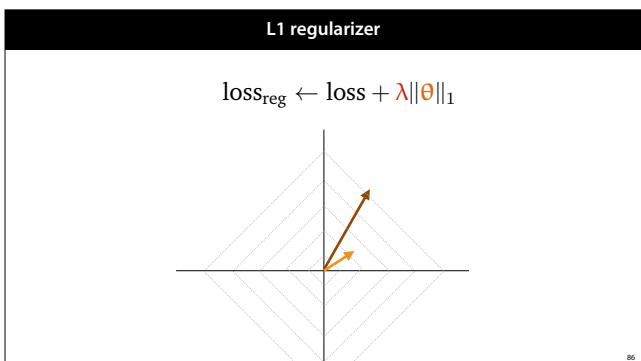
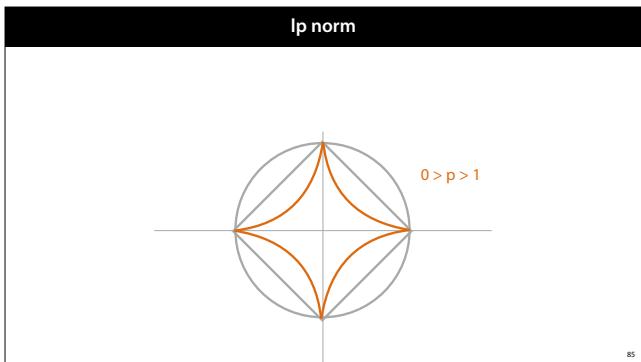
### $l_p$ norm

$$\|\theta\|^p = \sqrt[p]{w^p + b^p}$$



84

For the L1 norm, the form a diamond.



This means that for low  $p$  values ( $p=1$  is most common), we get

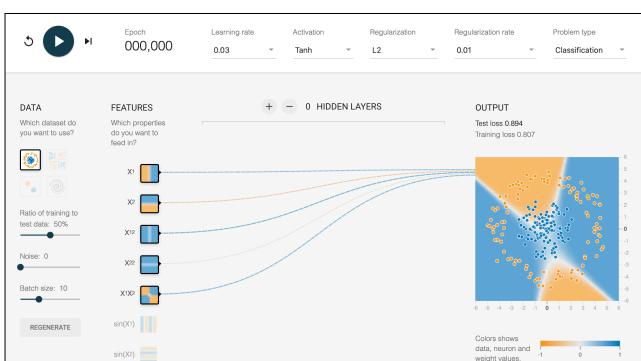
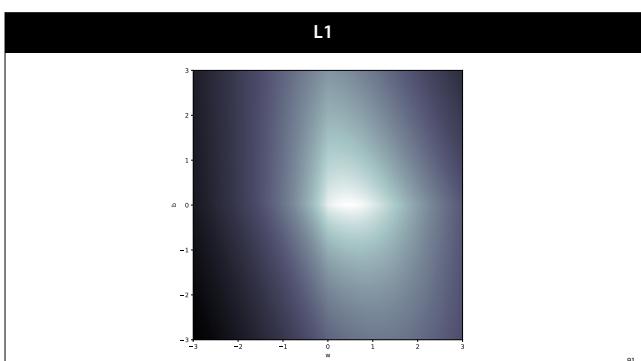
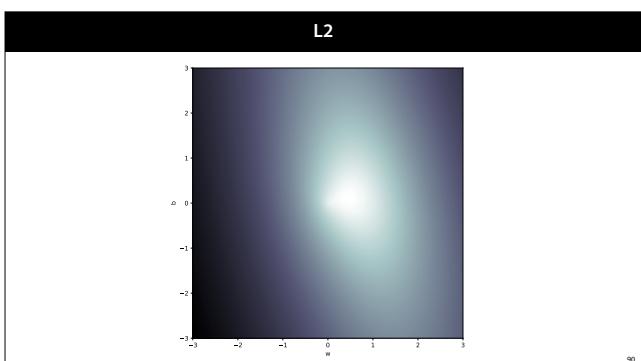
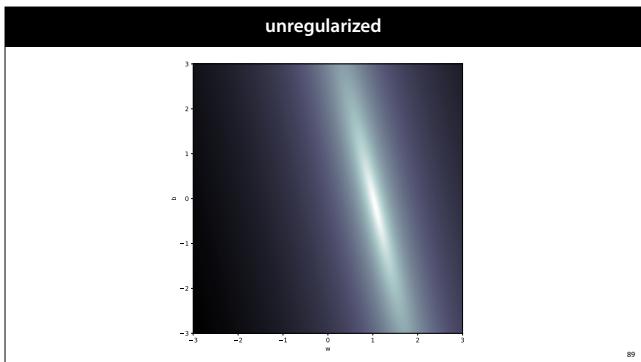
more sparse weight matrices: that is, if a weight is close to 0, it's more likely to get set entirely to zero.



Here's an analogy. Imagine you have a bowl, and you roll a marble down it to find the lowest point. Applying L2 loss is like tipping the bowl slightly to the right. You shift the lowest point in some direction (like to the origin).



L1 loss is like using a square bowl. It has grooves along the dimensions, so that when you tip the bowl, the marble is likely to end up in one of the grooves.



We can try this in tensor flow playground. For this example (a simple logistic regression) we know that the derived features  $x_1^2$  and  $x_2^2$  contain everything we need to a linear fit. However, when we with with regularly, or with L2 regularization, we see that the weights for the other features never quite go to zero. However, with L1 regularization, we see that they become precisely zero.

<https://bit.ly/32lc6cQ>

SVMs

**minimize :**

$$\frac{1}{2} \|\mathbf{w}\| + C \sum_i \max(0, 1 - y_i(\mathbf{w}^\top \mathbf{x}_i + b))$$

Sometimes a regularization term is somethin that you tack onto your model in an ad hoc fashion. Other times, it appears naturally. We saw this in the last lecture, where we rewrote the SVM soft margin loss to an error term and a regularization term.

Here the penalty hyperparameter **C** is on the error term, and not on the regularization term, but practically it doesn't make much of a difference which term you control.

## dropout

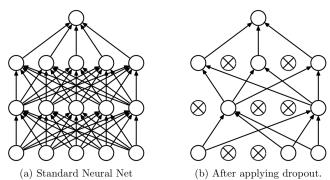


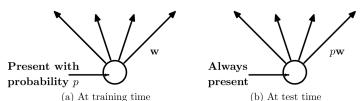
Figure 1: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Dropout is another regularization technique for large neural nets. During training, we simply remove hidden and input nodes (each with probability  $p$ ).

This prevents co-adaptation. Memorization (aka overfitting) often depends on multiple neurons firing together in specific combinations. Dropout prevents this.

image source: <http://jmlr.org/papers/v15/srivastava14a.html>

## dropout



**Figure 2: Left:** A unit at training time that is present with probability  $p$  and is connected to units in the next layer with weights  $w$ . **Right:** At test time, the unit is always present and the weights are multiplied by  $p$ . The output at test time is same as the expected output at training time.

Once you've finished training and you start using the model, you turn off dropout. Since this increases the size of the activations, you should correct by a factor of  $p$ .

Frameworks like Keras know when you're using the model to train and when you're using it to predict, and turn dropout on and off automatically.

## summary: regularization

Provides a soft preference for “simpler” models

| 2: Simpler means smaller parameters

**L1:** Simpler means smaller parameters

**Dropout:** randomly disable hidden units. Simpler means more robust

*Many other tricks available for regularization.*

**making it work**

How to make deep neural nets work.

- Overcoming vanishing gradients  
Proper initialisation, ReLUs over sigmoids
- Minibatch gradient descent
- Optimizers  
(Nesterov momentum, Adam)
- Regularizers  
L1/L2, Dropout

97

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

The Great A.I. Awakening

It's able to create knowledge itself: Google unveils AI that learns on its own

In a major breakthrough for artificial intelligence, AlphaGo Zero took just three days to master the ancient Chinese board game of Go ... with no human help

98

Once we had the basic frameworks for deep learning worked out and we started to get the hang of training big and deep networks, the deep learning revolution started to get going. Let's look at some early successes (mostly in the visual domain).

**natural language descriptions (2014)**

Describes without errors	Describes with minor errors	Somewhat related to the image	Related to the image

99

This is an end-to-end system for producing natural language descriptions of photographs. The system is not provided with any knowledge of the way language works, it just learns to produce captions from examples using a single neural network that consumes images and produces text, trained end-to-end.

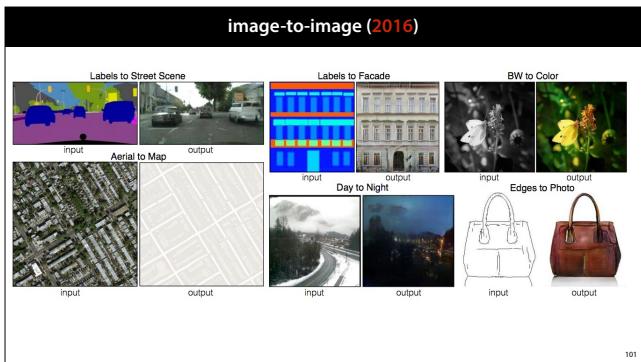
source: <https://www.engadget.com/2014/11/18/google-natural-language-image-description/>  
recommended: <https://twitter.com/picdeschot>

**Style transfer (2015)**

100

This example uses a convolutional network to transfer the style of one image onto another. Interestingly, this work was done with a general purpose network, trained on a general classification task (such networks are available for download). The authors took this network, and didn't change the weights. They just built the style transfer architecture around the existing network.

source: <https://research.googleblog.com/2016/10/supercharging-style-transfer.html>  
try it yourself: <http://demos.algorithmia.com/deep-style/>



This is pix2pix: a network with images as inputs and images as outputs was trained on various example datasets. Note the direction of the transformation. For instance, in the top left, the street scene with labeled objects was the *input*. The car-like objects, road surface, tree etc were all generated by the neural network to fill in the coloured patches in the input. Similarly, the bottom right shows the network generating a picture of a handbag *from a line-drawing*.



In some cases, we don't have neatly paired images: like the task of transforming a horse into a zebra. We can get a big bag of pictures of horses, and a big bag of pictures of zebras, but we don't know what a specific horse should look like as a zebra. The CycleGAN, published in 2017, could learn in this setting.

<http://gnotech.ae/news/cyclegan-ai-can-turn-classic-paintings-photos/>

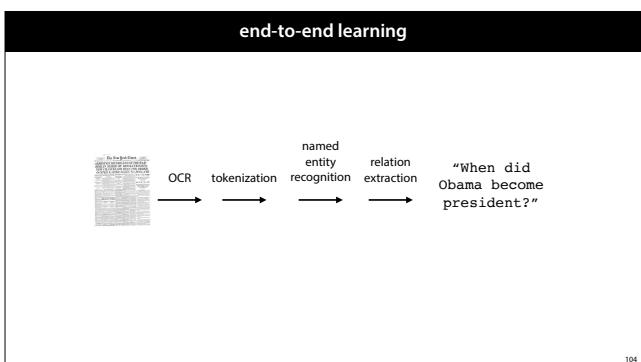
Ian Goodfellow  
@goodfellow\_ian

Following

4.5 years of GAN progress on face generation. [arxiv.org/abs/1406.2661](https://arxiv.org/abs/1406.2661) [arxiv.org/abs/1511.06434](https://arxiv.org/abs/1511.06434) [arxiv.org/abs/1606.07536](https://arxiv.org/abs/1606.07536) [arxiv.org/abs/1710.10196](https://arxiv.org/abs/1710.10196) [arxiv.org/abs/1812.04948](https://arxiv.org/abs/1812.04948)

2014      2015      2016      2017      2018

[https://twitter.com/goodfellow\\_ian/status/1084973596236144640](https://twitter.com/goodfellow_ian/status/1084973596236144640)



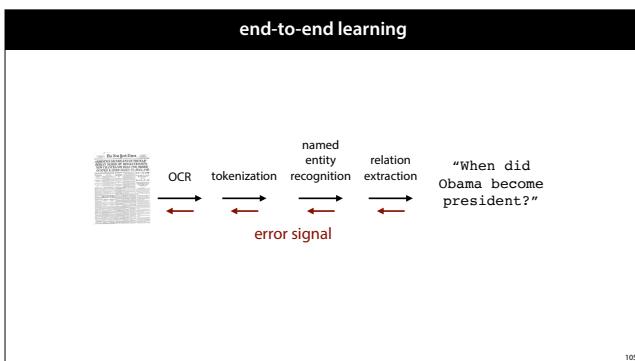
Finally, let's discuss what deep learning means on a higher level; why we consider it such a departure from classical machine learning.

Here is the kind of pipeline we would often attempt to build in the days before deep learning: we scan old news papers, perform optical character recognition, tokenise the characters into words, attempt to find named entities (like people and companies) and then try to learn the relations between these entities so that we can ask structured queries.

Most of these steps would be solved by some form of machine learning. And after a while, we were getting pretty good at each. So good that it would, for instance,

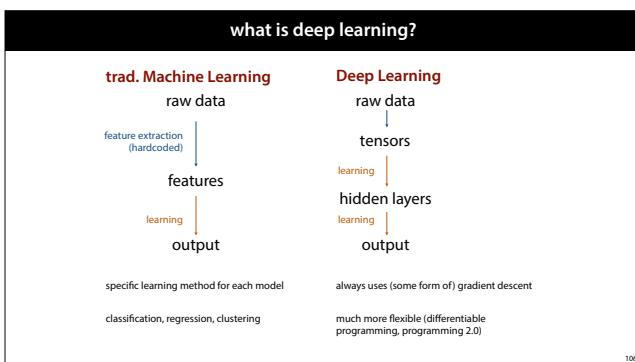
make a mistake for only 1 in a 100 instances.

But chaining together modules that are 99% accurate does not give you a pipeline that is 99% accurate. Error *accumulates*. The tokenization works slightly less well than on its pristine test data, because it's getting noisy input from the OCR. This makes the input for the NER module even more noisy and so on.



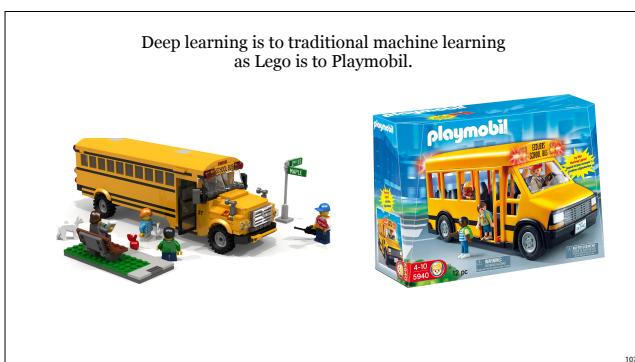
The solution of **deep learning** is to make each module differentiable: ensure that we can work out a local gradient so that we can also train the pipeline as a whole using backpropagation.

This is called **end-to-end learning**. Other names for roughly the same idea are **differentiable programming** and **programming 2.0**.



In traditional machine learning, the standard approach is to take our instances and to extract features. If our instances are things like images, natural language, or audio, this means we may lose information in this step. The data always has to be a matrix, so we are constrained to an inflexible abstract task.

In deep learning, because we translate our raw data to *tensors* of any shape and size, and then design a model to deal with the specific tensor shape we've created, we have much more flexibility, and we can get much closer to the raw data. This means that instead of deciding what the model should pay attention to through feature design, we are allowing the model to *learn* which aspects of the raw data are relevant.



In short, deep learning is to traditional machine learning, as Lego is to Playmobil. Both can give you a school bus, but the Lego school bus can be taken apart and reconstructed into a spaceship. The Playmobil bus is single-use.

These are different abstractions, with different purposes. Deep learning requires a little more work and insight, but you get a lot of flexibility in return.